Automated backward analysis of PKCS#11 v2.20

Robert Künnemann

Department of Computer Science – TU Darmstadt, Germany

Abstract. The PKCS#11 standard describes an API for cryptographic operations which is used in scenarios where cryptographic secrets need to be kept secret, even in case of server compromise. It is widely deployed and supported by many hardware security modules and smart cards. A variety of attacks in the literature illustrate the importance of a careful configuration, as API-level attacks may otherwise extract keys.

Formal verification of PKCS#11 configurations requires the analysis of a system that contains mutable state, a problem that existing methods solved by either artificially restricting the number of keys, introducing model-specific overapproximation or performing proofs by hand. At Security & Privacy 2014, Kremer and Künnemann presented a variant of the applied pi calculus that handles global state and, in conjunction with the tamarin prover for protocol verification, allows for the precise analysis of protocols with state. Using this tool chain, we show secrecy of keys for a PKCS#11 configuration that makes use of features introduced in version 2.20 of the standard, including wrap and unwrap templates in an extensible model.

This configuration supports the creation of so-called wrapping keys for import and export of sensitive keys (e. g., for backup or transfer), and it permits the coexistence of sensitive keys and non-sensitive keys on the same device.

1 Introduction

The more complex a system is, the more difficult is assuring its security. Given the complexity of the runtime environment provided by multi-purpose computers, it appears reasonable to compute security-critical operations outside the computer that actually runs the protocol (and usually more than one protocol), and instead on a device which is *a*) smaller, and thus more amenable to verification, and *b*) designed with security in mind. In security-critical contexts, e. g., the cash machine network or the public key infrastructure, the use of such devices, so-called *security tokens*, is common practice. In case the (complex) system running the protocol is under adversarial control, i. e., in case of server compromise, cryptographic secrets are protected by the (smaller, and thus more secure) security token, and the fact that these secrets were never directly accessible, even to the server.

PKCS#11 defines a platform-independent API to security tokens, for example smart cards, but also hardware security modules (HSMs). HSMs are physical computing devices that can be attached directly to a server via Ethernet, USB or other services, providing logical and physical protection for sensitive information and algorithms. The

fundamental security feature this standard provides is protecting the cryptographic values of keys marked as "sensitive" [22, Section 7]:

> Additional protection can be given to private keys and secret keys by marking them as "sensitive" [..]. Sensitive keys cannot be revealed in plaintext off the token [..].

Clulow's attack. It was discovered very soon that a faithful implementation of the standard violates this property. Clulow showed the following attack in 2003 [8], which serves as an introduction to the caveats in the design of PKCS#11. Keys are accessed via handles, hence indirectly. Some may be used for encryption and/or decryption, therefore an attacker that can access the token (e. g., in case of server compromise) can request the encryption of some message he supplies with a key of his choice, given that he knows the handle (which we will consider public) and that the key is configured to allow for encryption by setting an attribute 'encrypt' to true, analogously for decryption. Similarly, it is possible to *wrap* a key k_1 with another key k_2, that is, encrypt k_1 under k_2. This allows for backup, as well as for transfer. Given that k_2 is present on two security tokens A and B, one may wrap k_1 with k_2 on A (using two handles associated with k_1 and k_2, respectively) and *unwrap* the resulting cypher-text on B, using another handle to k_2. The scenario where k_2 is configured for wrapping *and* for decryption permits the attacker to request a wrapping of k_1 under k_2 and then decrypt the resulting cyphertext, thus obtaining the value of k_1 in the plain. The attack also applies when k_2 is wrapped under itself.

An 'incomplete' implementation of the standard, often called a *configuration* can thwart this attack by forbidding the same key to be used for wrapping and decryption. But there are other conflicts, e. g., between encryption and unwrapping. In the present paper, we propose a configuration and a method for verifying its security. We focus on attacks on the logical level, using only API calls that are (by themselves) perfectly harmless, as opposed to attacks on the implementation of cryptographic functionality [4].

Related Work. Building on the work of Longley and Rigby [20] and Bond and Anderson [5] on API attacks, several recent papers have investigated the security of APIs on the logical level adapting symbolic techniques for protocol analysis [6, 9, 11], finding many new attacks. There have also been academic proposals for new APIs [19, 10, 18]. While many attacks were found, a lot of effort was directed towards finding configurations that are secure, i. e., that preserve secrecy of keys.

In the analysis of PKCS#11 configurations, there are three major lines of work. The first one uses protocol verification techniques, regarding the security token as the (sole) participant in a protocol, with the adversary sending requests on the public network. Early results by Delaune, Kremer and Steel translated a given configuration into a satisfiability problem which is solved by model checking, providing secrecy of keys if the number of keys is known in advance [11]. This restriction was lifted in later work [14] by Fröschle and Steel, showing that a class of configurations can safely be abstracted by configurations that are *static* (i. e., a key's attribute cannot be changed) and showing that the latter is soundly over-approximated in a bounded model. This method is used in further work by Bortolozzo, Centenaro, Focardi and Steel to find attacks on security tokens using automatic reverse engineering and to show a configuration very similar to ours secure.

The second line of work uses a program verification approach, modelling security tokens in a first-order linear time logic with past operators. In contrast to the protocol verification approach, proofs have been conducted by hand (using a tableau proof method that proceeds by backward analysis), but provide support for advanced data structures introduced by PKCS#11, version 2.20, in particular wrap/unwrap templates. An *attribute template* is a set of attributes, but contains the attributes 'wrapping template' and 'unwrapping template' which themselves are pointers to attribute templates, resulting in a recursive data structure. In early work, Fröschele and Sommer show security, but only for keys with 'extractable' set to false [15], i. e., keys that cannot be wrapped at all. In more recent work, the authors added support for wrap/unwrap templates [13]. Positives result only cover the secrecy of trusted wrapping keys, but not of keys wrapped using these keys (as opposed to the results in the present paper, and in the work by Bortolozzo et al.). The proof is done by hand and covers about six pages [16].

The third line of work uses static analysis on the implementation of the security token [7]. Using security type-checking, a software implementation (written in a subset of C) of the wrap, unwrap, encrypt and decrypt functions was shown to preserve secrecy of sensitive keys against a Dolev-Yao attacker for a configuration that is functionally equal to the one presented here, but uses a default type for wrapped keys rather than wrap/unwrap templates. A generalised version of this framework proposes an imperative programming language with cryptographic operations and a centralised store mapping values (possibly handles) to pairs of key values and their ground types [2]. Mapping sets of PKCS#11 attributes to types, the generalised framework again allows for the analysis of PKCS#11, more specifically, PKCS#11 v2.20 using the configuration analysed in the present paper.

Contribution. The present work shares common ground with the first two lines of work. It follows a protocol verification approach and provides machine support, but relies on backward analysis rather than finite model checking. Therefore, the resulting model can be extended without breaking the soundness of the decision procedure (or introducing limitations to the number of keys), but provides a largely automated proof procedure although decidability cannot be guaranteed. The model is formulated in a variant of the applied pi calculus augmented with operators for state manipulation. This high-level protocol description language can be translated to multiset rewrite rules using the Sapic tool which has been proven sound and complete in prior work [17]. The generated multiset rewrite system can be verified using the tamarin prover [23], which is sound and complete as well. The constraint solving algorithm employed by the tamarin prover uses backward analysis. While various methods in the first line of work achieve security for an unbounded number of keys indirectly (using bounded analysis and an over-approximation specific to the modelling of the PKCS#11 API) [14, 6], this method supports an unlimited number of fresh values by default. For one, our model is closer to an actual implementation, containing locking operations as well as database lookup, but most importantly, our result requires no additional over-approximation. Hence, the security proof is machine-verifiable (relying on the correctness of the translation procedure [17] and the solving algorithm [23]) and the model extensible. The same holds for the third line of work, too (except that parallelism and hence locking is not supported). The type-checking approach is much faster and requires less manual intervention than

our approach, but might not be able to type configurations that are actually secure. However, the configuration we show secure has been verified using type-checking [7, 2], hence the question of whether there actually is a more versatile policy that can be shown secure using backward analysis but is not amenable to security type-checking remains open.

The flexibility and precision of our modelling comes at the cost of losing some amount of automation. As the verification method is sound and complete, but is able to express the secrecy problem with unbounded nonces, which is undecidable [12], there is no guarantee that the tool terminates. In order to achieve termination, some intervention is necessary: Lemmas need to be used to cut branches in the proof attempt. The advantage is a result that applies to a version of PKCS#11 (2.20) with wrap/unwrap templates and an intuitive, extensible model. In the present proof, we have given 10 lemmas, 6 of which are at least partially guided when being used without any model-specific heuristics for the choice of goals in the proof. If we provide a (model-specific) heuristic, a proof is found without user intervention. The lemmas are model-specific, but not specific to the configuration we show secure (except for four trivial lemmas that speed up the proof, but could be left out). We note that the tool chain is sound and complete no matter which heuristic is used, i. e., even when the model is altered (e. g., to describe future versions of PKCS#11), if the heuristics are helpful in finding a proof, the result is correct.

The configuration of PKCS#11 we prove secure implements a policy with three kinds of keys: keys that are used to encrypt or decrypt payload and kept secret, keys that can wrap and unwrap the first kind of keys for backup and keys that can be read in the clear and are neither wrapped, nor used to wrap other keys but may encrypt/decrypt payload. Keys cannot change roles, which we consider a sensible best practice, thus this policy is static in the sense of [14].

This policy has been shown to provide secrecy of sensitive keys if implemented via restricting the wrap and unwrap commands [7] or via wrap/unwrap templates [2] using type-checking. As in these works, we show that non-sensitive keys (keys the attacker is allowed to read in the clear) do not impair the security of other keys.

2 The PKCS#11 standard, v2.20

The PKCS#11 standard specifies a security API (this term was coined by Bond and Anderson [5]), that is, an API that separates trusted code operating on secret data from untrusted code. As in the case of PKCS#11, security APIs are often used to perform cryptographic computations. Separating the implementation of cryptographic operations from the rest of the system has the advantage that cryptographic secrets can be hidden *behind* the API, whilst the code only accesses these secrets indirectly, using handles. If the API gives access to an external piece of hardware, which is often the case, the hope is that malicious code is restricted to using the API. Smart cards and hardware security modules (HSMs) implementing PKCS#11 are much simpler than multi-purpose computers, and designed with security in mind, so the idea of "outsourcing" sensitive information and operations that depend on this information to a device

name	modifiable	SO-only	comment
wrap (`wrap`)	yes	no	can be first argument for wrap
unwrap (`unwrap`)	yes	no	can be first argument for unwrap
encrypt (`enc`)	yes	no	can be used for encryption
decrypt (`dec`)	yes	no	can be used for decryption
sensitive (`sens`)	yes	no	value shall not be extracted (directly), but may be wrapped
extractable (`extr`)	yes	no	Value shall not be extracted (directly) or used as second argument to wrap
trusted (`trus`)	no	yes	has been generated by SO
wrap-with-trusted (`wwt`)	no	no	can only be wrapped by 'trusted' keys
wrap template (`wt`)	no	no	key that are wrapped need to match this template
unwrap template (`ut`)	no	no	template applied to keys after being imported by unwrap

Table 1: Attributes relevant to key-management. Modifiable means that attributes may be modified after an object is created, or while it is copied, however, tokens may decide not to permit modification upon copying [22, Tab. 15, footnote 8, p. 66]. SO-only means the attribute can only be set by the SO.

that is easier to secure seems to be reasonable. However, a sound design of said API is essential in reaching this goal.

PKCS#11 gives multiple applications access to several cryptographic devices via *slots* [22, p.14], abstracting away from their respective implementation technology, be it smart cards, PCMCIA cards, HSMs, etc. After a user, or an application, has established a *session* to a device (through a slot), they can identify either as a Security Officer (SO), or a normal user [22, p.15]. The role of the SO is to initialize a token and to set the normal user's PIN. The normal user cannot log in until the SO has set the normal user's PIN. Within a session, the user can manipulate *objects* stored on the token, such as keys and certificates. Objects are referred to by *handles*. The value of a handle does not reveal any information about the value of the key. Objects may be marked public and private. A normal user, if authenticated, can access public and private objects, otherwise only public objects. The SO can only access public objects, but perform operations the normal user cannot perform, such as setting the user's PIN.

Attributes Objects may have attributes (besides being public or private), some of them specific to their class and type (public keys of type `CKK_RSA` have a public exponent, for example), and some general. The latter pertain to the key-management functions of PKCS#11, and are listed in Table 1. Real devices offer only a subset of the functionality specified by PKCS#11, partly due to security considerations: The PKCS#11 standards permits modifying the attributes wrap and dec, immediately giving rise to Clulow's attack, see p.2.

2.1 Modelling

Security APIs are a means to provide a higher level of assurance in case of server compromise. If the server is compromised, the user PIN can easily be intercepted and the attacker can establish a session with the device. Hence we model a network comprising only of a single PKCS#11 token, which does not perform any kind of authentication for normal users. The Dolev-Yao adversary is in full control over the network and is thus able to issue arbitrary requests. The PKCS#11 standard discusses security against server compromise and states that "none of the [these attacks] can compromise keys marked 'sensitive', since a key that is sensitive will always remain sensitive" [22, p. 31]. This does not hold true for PKCS#11 itself (cf. p. 2), but we will show a configuration of PKCS#11, for which this holds true.

This paper describes a formal model of the core key-management functionality in PKCS#11, therefore we left out message digesting functions, signature and MAC functions, as well as dual-purpose cryptographic functions (like C_DigestEncryptUpdate) and random number generation, as they do not pertain to the key and object management part of PKCS#11. Key-derivation (C_DeriveKey) allows for creating a new key object from a base key and could be considered key-management, but there is no cryptographic mechanism in PKCS#11 that supports wrap/unwrap as well as key derivation [22, Section 12]. Thus we consider this function outside the key-management core of PKCS#11. Note that encryption and decryption functions may allow for producing, or decrypting wrappings (as Clulow's attack shows), so they do pertain to the key-management part. We concentrate on encryption and decryption for single-part data, i. e., the functions for multi-part data C_EncryptInit, C_EncryptUpdate, C_EncryptFinal are left out in favour of C_Encrypt, as our focus lies on API-level attacks. The same holds for decryption functions. We furthermore ignored object management functions that reveal only information which the attacker, as the only user accessing the token, can compute anyway, such as C_GetObjectSize, C_GetAttributeValue, C_FindObjectsInit, C_-FindObjects and C_FindObjectsFinal. The function C_DestroyObject is disabled, since our modelling gives the adversary an unlimited amount of space to store keys, thus any attack can be transformed into an attack that does not delete objects. Previous works [11, 14, 6, 15] have similar restrictions. The full list of functions is listed in Table 8 of the PKCS#11 standard [22, p. 27]. The rest of PKCS#11 is what we consider the core key-management part (Table 2). The configuration, while being more versatile than policies that were previously proposed, forbids certain operations that are potentially harmful. Another class of attacks Clulow discovered are "Trojan wrapped key attacks", where a public unwrapping key is used to introduce a wrapping key that the attacker knows the value of, by producing the wrapping outside the device. We conclude that our policy should not allow for asymmetric wrapping keys at all, and simplify the model by only regarding symmetric keys. Extending the model to cover asymmetric keys for non-key-management operations is straight-forward, but unlikely to lead to new insights with respect to the security of policies.

function	description	process	comment
Object management functions			
C_CreateObject	creates an object	(n.p.)	forbidden by policy
C_CopyObject	creates a copy of an object	(n.p.)	not helpful to adversary (in this configuration)
C_DestroyObject	destroys an object	(n.p.)	not helpful to adversary
C_SetAttributeValue	modifies object's attribute	(n.p.)	forbidden by policy
Encryption/Decryption functions			
C_Encrypt	encrypts single-part data	encrypt	
C_Decrypt	decrypts single-part data	decrypt	
Key-management functions			
C_GenerateKey	generates a secret key	create	
C_GenerateKeyPair	generates a public / private key pair	(n.p.)	asymm. wrapping keys permits 'Trojan wrapped key attack'
C_WrapKey	wraps (encrypts) a key	wrap	
C_UnwrapKey	unwraps (decrypts) a key	unwrap	

Table 2: Object- and Key-management operations in PKCS#11 and our modelling. Function marked '(n.p.)' are not present.

2.2 Proposed configuration

In this section, we will discuss policies that have been proposed in previous work and introduce requirements that guided the design of the policy we will prove secure. We will argue that, given these requirements, there are only two secure policies.

The first requirement shall be that there are 'usage' keys, i. e., keys that may encrypt or sign payload. Since only encryption and decryption are relevant to key-management, we propose:

> Requirement A: There should be keys that can be used for encryption and decryption.

Delaune et al. have shown different policies secure, all of which have in common that wrap and unwrap are conflicting attributes [11]. Since wrapping keys are useful for backup and out-of-device-storage of keys, we could formulate the requirement that wrapping keys should be suitable for wrapping and unwrapping any key, however, we think that the backup, and possibly the distribution of 'usage' keys – in our model, keys used for encryption and decryption – is the main purpose of wrapping keys. Being able to wrap wrapping keys is useful, but only once it is established that usage keys can be wrapped.

> Requirement B: Wrapping keys should be able to wrap and unwrap keys that can be used for encryption and decryption.

Policies in related work often support the setting and unsetting of some attributes [15, 13], requiring attributes to be unmodifiable where needed. Many configurations describe policies by declaring conflicting attributes (pairs of attributes that are forbidden

to be set at once) and sticky (unmodifiable when set) attributes. We argue for a different approach: security policies become much easier to understand and to design if the key is assigned a role upon creation, and cannot be altered in subsequent steps. In most cases, it is clear from the beginning which keys needs to be protected, setting a key to sensitive after it was ready to be exposed is clearly bad practice. The attributes wrap, unwrap, enc, dec, extr, trust have in common that an attacker becomes only less powerful in unsetting them. The opposite holds for wwt. Being able to alter wrapping/unwrapping templates is much less useful as it seems, as wrappings in PKCS#11 carry no information about which wrapping template was used. Altering the wrapping template could be useful to define the class of keys that can be wrapped as the union of permitted wrapping templates but, as we will see, this class can be defined using a single wrapping template.

Requirement C: Policies should disable C_SetAttributeValue altogether.

Bortolozzo et al. proposed a policy that is static in that attributes cannot be altered after the creation of keys [6]. Ignoring templates where wrap, unwrap, enc and dec are false, they propose three templates:

1. Freshly generated keys can have wrap and unwrap set.
2. Freshly generated keys can have enc and dec set.
3. Keys imported with unwrap may have unwrap and encrypt set, but wrap and decrypt unset.

Intuitively, the third template means that keys 'degrade' when being backed up, that is, they cannot resume to their full functionality. The policy we analyse here differs in this regard: 'trusted' keys can be used to wrap 'usage' keys, which can be adequately restored.

name	wrap	unwrap	enc	dec	sens	extr	trus	wwt	wt	ut
trusted	•	•			•	•	•	•	usage	usage
usage			•	•	•	•		•	-	-
untrusted			•	•		•			-	-

Table 3: The templates using in the proposed policy. (A dot • is present for each attribute that is set.)

These policies are incomparable; while the second policy supports lossless key-backup, the policy by Bortolozzo et al. allows for transferring wrapping keys between two PKCS#11 device A and B. Assume A and B have the same wrapping key set-up at the start (e. g., by the SO). A creates another wrapping key, which is wrapped using the common wrapping key, and unwrapped at B. Now A can use this second-level wrapping key to produce wrappings that B can import.

3 Preliminaries

Terms and equational theories As usual in symbolic protocol analysis we model messages by abstract terms. Therefore we define an order-sorted term algebra with the sort

msg and two incomparable subsorts pub and $fresh$. For each of these subsorts we assume a countably infinite set of names, FN for fresh names and PN for public names. Fresh names will be used to model cryptographic keys and nonces while public names model publicly known values. We furthermore assume a countably infinite set of variables for each sort s, $\mathcal{V}_s$ and let $\mathcal{V}$ be the union of the set of variables for all sorts. We write $u : s$ when the name or variable u is of sort s. Let Σ be a signature, i.e., a set of function symbols, each with an arity. We write f/n when function symbol f is of arity n. We denote by $\mathcal{T}_\Sigma$ the set of well-sorted terms built over Σ, PN, FN and $\mathcal{V}$. For a term t we denote by $names(t)$, respectively $vars(t)$ the set of names, respectively variables, appearing in t. The set of ground terms, i.e., terms without variables, is denoted by $\mathcal{M}_\Sigma$. When Σ is fixed or clear from the context we often omit it and simply write $\mathcal{T}$ for $\mathcal{T}_\Sigma$ and $\mathcal{M}$ for $\mathcal{M}_\Sigma$.

We equip the term algebra with an equational theory E, that is, a finite set of equations of the form $M = N$ where $M, N \in \mathcal{T}$. From the equational theory we define the binary relation $=_E$ on terms, which is the smallest equivalence relation containing equations in E that is closed under application of function symbols, bijective renaming of names and substitution of variables by terms of the same sort. Furthermore, we require E to distinguish different fresh names, i. e., $\forall a, b \in FN : a \neq b \Rightarrow a \neq_E b$.

Example 1. Symmetric encryption can be modelled using a signature

$$\Sigma = \{\, senc/2, sdec/2, true/0 \,\}$$

and an equational theory defined by

$$sdec(senc(m, k), k) = m.$$

For the rest of the paper we assume that E refers to some fixed equational theory and that the signature and equational theory always contain symbols and equations for pairing and projection, i.e., $\{\langle .,. \rangle, \mathsf{fst}, \mathsf{snd}\} \subseteq \Sigma$ and equations $\mathsf{fst}(\langle x, y\rangle) = x$ and $\mathsf{snd}(\langle x, y\rangle) = y$ are in E. We will sometimes use $\langle x_1, x_2, \ldots, x_n\rangle$ as a shortcut for $\langle x_1, \langle x_2, \langle \ldots, \langle x_{n-1}, x_n\rangle \ldots\rangle$.

We also use the usual notion of positions for terms. A position p is a sequence of positive integers and $t|_p$ denotes the subterm of t at position p.

Facts We also assume an unsorted signature Σ_{fact}, disjoint from Σ. The set of *facts* is defined as

$$\mathcal{F} := \{F(t_1, \ldots, t_k) \mid t_i \in \mathcal{T}_\Sigma, F \in \Sigma_{fact} \text{ of arity } k\}.$$

Facts will be used both to annotate protocols, by the means of events, and for defining multiset rewrite rules. We partition the signature Σ_{fact} into *linear* and *persistent* fact symbols. We suppose that Σ_{fact} always contains a unary, persistent symbol $!\mathsf{K}$ and a linear, unary symbol Fr. Given a sequence or set of facts S we denote by $lfacts(S)$ the multiset of all linear facts in S and $pfacts(S)$ the set of all persistent facts in S. By notational convention facts whose identifier starts with '!' will be persistent. $\mathcal{G}$ denotes the set of ground facts, i.e., the set of facts that do not contain variables. For a fact f we denote by $ginsts(f)$ the set of ground instances of f. This notation is also lifted to sequences and sets of facts as expected.

Predicates We assume an unsorted signature Σ_{pred} disjoint from Σ and Σ_{fact}. The set of *predicates* is defined as

$$\mathcal{P} := \{pr(t_1, \ldots, t_k) \mid t_i \in \mathcal{T}_\Sigma, pr \in \Sigma_{pred} \text{ of arity } k\}.$$

Predicates will be used to describe branching conditions in protocols. Each predicate is defined via a first-order formula over ground atoms of the form $t_1 \approx t_2$, i.e., the grammar for such formulae is

$$\langle \phi \rangle ::= t_1 \approx t_2 \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \exists x.\phi$$

where t_1, t_2 are terms and $x \in \mathcal{V}$. For an n-ary predicate pr, $pr(x_1, ..., x_n)$ is defined by a formula ϕ_{pr} such that $fv(\phi_p) \subseteq x_1, ..., x_n$, where fv denotes the free variables in a formula, i. e., variables $v \in \mathcal{V}$ not bound by $\exists v$. The semantics of the first-order formulae is as usual where we interpret $\approx$ as $=_E$. We use $\sigma_1 \vee \sigma_2$ as short-hand for $\neg(\neg\sigma_1 \wedge \neg\sigma_2)$.

Example 2. A predicate $pr = can\_wrap$ is used to check whether the attributes associated to two handles (10 attributes each) allow for wrapping. For readability, we rename x_1 to $wrap_1$, x_{15} to $extr_2$, x_7 to $trus_1$ and x_{18} to wwt_2:

$$\begin{aligned}\sigma_{can\_wrap} :=& (wrap_1 \approx \text{'on'} \wedge extr_2 \approx \text{'on'}) \wedge \\ & ((wwt2 \approx \text{'off'}) \vee (wwt_2 \approx \text{'on'} \wedge trus_1 \approx \text{'on'}))\end{aligned}$$

Substitutions A substitution σ is a partial function from variables to terms. We suppose that substitutions are well-typed, i.e., they only map variables of sort s to terms of sort s, or of a subsort of s. We denote by $\sigma = \{^{t_1}/_{x_1}, \ldots, ^{t_n}/_{x_n}\}$ the substitution whose domain is $\mathbf{D}(\sigma) = \{x_1, \ldots, x_n\}$ and which maps x_i to t_i. As usual we homomorphically extend σ to apply to terms and facts and use a postfix notation to denote its application, e.g., we write $t\sigma$ for the application of σ to the term t. A substitution σ is grounding for a term t if $t\sigma$ is ground. Given function g we let $g(x) = \perp$ when $x \notin \mathbf{D}(x)$. When $g(x) = \perp$ we say that g is undefined for x. We define the function $f := g[a \mapsto b]$ with $\mathbf{D}(f) = \mathbf{D}(g) \cup \{\, a \,\}$ as $f(a) := b$ and $f(x) := g(x)$ for $x \neq a$.

Sets, sequences and multisets We write $\mathbb{N}_n$ for the set $\{1, \ldots, n\}$. Given a set S we denote by S^* the set of finite sequences of elements from S and by $S^\#$ the set of finite multisets of elements from S. We use the superscript $^\#$ to annotate usual multiset operations, e. g., $S_1 \cup^\# S_2$ denotes the multiset union of multisets S_1, S_2. Given a multiset S we denote by $set(S)$ the set of elements in S. The sequence consisting of elements $e_1, \ldots, e_n$ will be denoted by $[e_1, \ldots, e_n]$ and the empty sequence is denoted by $[\,]$. Given a sequence S, we denote by $idx(S)$ the set of positions in S, i.e., $\mathbb{N}_n$ when S has n elements, and for $i \in idx(S)$ S_i denotes the ith element of the sequence. Set membership modulo E is denoted by $\in_E$ and defined as $e \in_E S$ if $\exists e' \in S.\ e' =_E e$. $\subset_E$ and $=_E$ are defined for sets in a similar way. Application of substitutions is lifted to sets, sequences and multisets as expected. By abuse of notation we sometimes interpret sequences as sets or multisets; the applied operators should make the implicit cast clear.

$\langle P,Q \rangle$ ::= 0
| $P \mid Q$
| $!\,P$
| νn: *fresh*; P
| out(M, N); P
| in(M, N); P
| if $Pred$ then P [else Q] $\quad Pred \in \mathcal{P}$
| event F ; P $\quad (F \in \mathcal{F})$

$\langle P,Q \rangle$ ::= *(continued)*
| insert M,N; P
| delete M; P
| lookup M as x in P [else Q]
| lock M; P
| unlock M; P
| $[L] \dashv[\ A]\mapsto [R]; P \quad (L, R, A \in \mathcal{F}^*)$

Fig. 1: Syntax, where $M, N \in \mathcal{T}$

3.1 A cryptographic pi calculus with explicit state

Syntax and informal semantics The Sapic calculus is a variant of the applied pi calculus [1]. In addition to the usual operators for concurrency, replication, communication and name creation, it offers several constructs for reading and updating an explicit global state. The grammar for processes is described in Fig. 1.

0 denotes the terminal process. $P \mid Q$ is the parallel execution of processes P and Q and $!P$ the replication of P, allowing an unbounded number of sessions in protocol executions. The construct $\nu n; P$ binds the name $n \in FN$ in P and models the generation of a fresh, random value. Processes out(M, N); P and in(M, N); P represent the output, respectively input, of message N on channel M. Readers familiar with the applied pi calculus [1] may note that we opted for the possibility of pattern matching in the input construct, rather than merely binding the input to a variable x. The process if $Pred$ then P else Q will execute P or Q, depending on whether $Pred$ holds. For example, if $Pred = equal(M, N)$, and $\phi_{equal} = x_1 \approx x_2$, then if $equal(M, N)$ then P else Q will execute P if $M =_E N$ and Q otherwise. (In the following, we will use $M = N$ as short-hand for $equal(M, N)$.) The event construct is merely used for annotating processes and will be useful for stating security properties. For readability we sometimes omit to write else Q when Q is 0, as well as trailing 0 processes.

The remaining constructs are used for manipulating state and are new compared to the applied pi calculus. We offer two different mechanisms for state. The first construct is *functional* and allows to associate a value to a key. The construct insert M,N binds the value N to a key M. Successive inserts allow to change this binding. The delete M operation simply "undefines" the mapping for the key M. The lookup M as x in P else Q allows to retrieve the value associated to M, binding it to the variable x in P. If the mapping is undefined for M the process behaves as Q. The lock and unlock constructs allow to gain exclusive access to a resource M. This is essential for writing protocols where parallel processes may read and update a common memory. We additionally offer another kind of global state in form of a multiset of ground facts, as opposed to the previously introduced functional store. The purpose of this construct is to provide access to the underlying notion of state in tamarin, but we stress that it is distinct from the previously introduced functional state, and its use is only advised to expert users. It is not used in the present modelling.

The bound names of a process are those that are bound by νn. We suppose that all names of sort *fresh* appearing in the process are under the scope of such a binder.

Free names must be of sort pub. A variable x can be bound in three ways: *(i)* by the construct lookup M as x, or *(ii)* $x \in vars(N)$ in the construct in(M, N) and x is not under the scope of a previous binder, *(iii)* $x \in vars(L)$ in the construct $[L] \multimap [\ A]\!\rightarrow [R]$ and x is not under the scope of a previous binder. While the construct lookup M as x always acts as a binder, the input and $[L] \multimap [\ A]\!\rightarrow [R]$ constructs do not rebind an already bound variable but perform pattern matching.

A process is ground if it does not contain any free variables. We denote by $P\sigma$ the application of the homomorphic extension of the substitution σ to P. As usual we suppose that the substitution only applies to free variables. We sometimes interpret the syntax tree of a process as a term and write $P|_p$ to refer to the subprocess of P at position p (where $|$, if and lookup are interpreted as binary symbols, all other constructs as unary).

Semantics

Frames and deduction Before giving the formal semantics of our calculus we introduce the notions of frame and deduction. A *frame* consists of a set of fresh names $\tilde{n}$ and a substitution σ and is written $\nu\tilde{n}.\sigma$. Intuitively a frame represents the sequence of messages that have been observed by an adversary during a protocol execution and secrets $\tilde{n}$ generated by the protocol, a priori unknown to the adversary. Deduction models the capacity of the adversary to compute new messages from the observed ones.

Definition 1 (Deduction). *We define the deduction relation $\nu\tilde{n}.\sigma \vdash t$ as the smallest relation between frames and terms defined by the deduction rules in Figure 2.*

$$\frac{a \in FN \cup PN \quad a \notin \tilde{n}}{\nu\tilde{n}.\sigma \vdash a}\ \textsc{Dname} \qquad \frac{\nu\tilde{n}.\sigma \vdash t \quad t =_E t'}{\nu\tilde{n}.\sigma \vdash t'}\ \textsc{DEq}$$

$$\frac{x \in \mathbf{D}(\sigma)}{\nu\tilde{n}.\sigma \vdash x\sigma}\ \textsc{DFrame} \qquad \frac{\nu\tilde{n}.\sigma \vdash t_1 \cdots \nu\tilde{n}.\sigma \vdash t_n \quad f \in \Sigma^k}{\nu\tilde{n}.\sigma \vdash f(t_1, \ldots, t_n)}\ \textsc{DAppl}$$

Fig. 2: Deduction rules.

Operational semantics We can now define the operational semantics of our calculus. The semantics is defined by a labelled transition relation between process configurations. A *process configuration* is a 6-tuple $(\mathcal{E}, \mathcal{S}, \mathcal{S}^{\mathrm{MS}}, \mathcal{P}, \sigma, \mathcal{L})$ where

- $\mathcal{E} \subseteq FN$ is the set of fresh names generated by the processes;
- $\mathcal{S} : \mathcal{M}_\Sigma \to \mathcal{M}_\Sigma$ is a partial function modeling the functional store;
- $\mathcal{S}^{\mathrm{MS}} \subseteq \mathcal{G}^{\#}$ is a multiset of ground facts and models the multiset of stored facts;
- $\mathcal{P}$ is a multiset of ground processes representing the processes executed in parallel;
- σ is a ground substitution modeling the messages output to the environment;
- $\mathcal{L} \subseteq \mathcal{M}_\Sigma$ is the set of currently acquired locks.

The transition relation is defined by the rules described in Fig. 3. Transitions are labelled by sets of ground facts. For readability we omit empty sets and brackets around singletons, i.e., we write $\rightarrow$ for $\xrightarrow{\emptyset}$ and $\xrightarrow{f}$ for $\xrightarrow{\{f\}}$. We write $\rightarrow^*$ for the reflexive, transitive closure of $\rightarrow$ (the transitions that are labelled by the empty sets) and write $\stackrel{f}{\Rightarrow}$ for $\rightarrow^* \xrightarrow{f} \rightarrow^*$. We can now define the set of traces, i.e., possible executions, that a process admits.

Definition 2 (Traces of P). *Given a ground process P we define the* set of traces of P *as*

$$traces^{pi}(P) = \Big\{ [F_1, \ldots, F_n] \mid (\emptyset,\emptyset,\emptyset,\{P\},\emptyset,\emptyset) \stackrel{F_1}{\Longrightarrow} (\mathcal{E}_1,\mathcal{S}_1,\mathcal{S}_1^{\mathrm{MS}},\mathcal{P}_1,\sigma_1,\mathcal{L}_1) \stackrel{F_2}{\Longrightarrow} \ldots \stackrel{F_n}{\Longrightarrow} (\mathcal{E}_n,\mathcal{S}_n,\mathcal{S}_n^{\mathrm{MS}},\mathcal{P}_n,\sigma_n,\mathcal{L}_n) \Big\}$$

3.2 Security Properties

The formalism used for defining security properties in the Sapic tool, which is used to define key secrecy as well as helping lemmas was introduced with the tamarin tool [23]. Security properties are described in an expressive two-sorted first-order logic. The sort $temp$ is used for time points, $\mathcal{V}_{temp}$ are temporal variables.

Definition 3 (Trace formulas). *A trace atom is either false $\bot$, a term equality $t_1 \approx t_2$, a timepoint ordering $i \lessdot j$, a timepoint equality $i \doteq j$, or an action $F@i$ for a fact $F \in \mathcal{F}$ and a timepoint i. A trace formula is a first-order formula over trace atoms.*

As we will see in our case studies this logic is expressive enough to analyze a variety of security properties, including complex injective correspondence properties.

To define the semantics, let each sort s have a domain $\mathbf{D}(s)$. $\mathbf{D}(temp) = \mathcal{Q}$, $\mathbf{D}(msg) = \mathcal{M}$, $\mathbf{D}(fresh) = FN$, and $\mathbf{D}(pub) = PN$. A function $\theta : \mathcal{V} \rightarrow \mathcal{M} \cup \mathcal{Q}$ is a valuation if it respects sorts, that is, $\theta(\mathcal{V}_s) \subset \mathbf{D}(s)$ for all sorts s. If t is a term, $t\theta$ is the application of the homomorphic extension of θ to t.

Definition 4 (Satisfaction relation). *The satisfaction relation $(tr,\theta) \vDash \varphi$ between trace tr, valuation θ and trace formula φ is defined as follows:*

$$\begin{array}{ll}
(tr,\theta) \vDash \bot & \textit{never} \\
(tr,\theta) \vDash F@i & \textit{iff } \theta(i) \in idx(tr) \textit{ and } F\theta \in_E tr_{\theta(i)} \\
(tr,\theta) \vDash i \lessdot j & \textit{iff } \theta(i) < \theta(j) \\
(tr,\theta) \vDash i \doteq j & \textit{iff } \theta(i) = \theta(j) \\
(tr,\theta) \vDash t_1 \approx t_2 & \textit{iff } t_1\theta =_E t_2\theta \\
(tr,\theta) \vDash \neg\varphi & \textit{iff not } (tr,\theta) \vDash \varphi \\
(tr,\theta) \vDash \varphi_1 \wedge \varphi_2 & \textit{iff } (tr,\theta) \vDash \varphi_1 \textit{ and } (tr,\theta) \vDash \varphi_2 \\
(tr,\theta) \vDash \exists x : s.\varphi & \textit{iff there is } u \in \mathbf{D}(s) \textit{ such that } (tr,\theta[x \mapsto u]) \vDash \varphi
\end{array}$$

When φ is a ground formula we sometimes simply write $tr \vDash \varphi$ as the satisfaction of φ is independent of the valuation.

Standard operations:

$$(\mathcal{E},\mathcal{S},\mathcal{S}^{\mathrm{MS}},\mathcal{P}\cup^{\#}\{0\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E},\mathcal{S},\mathcal{S}^{\mathrm{MS}},\mathcal{P},\sigma,\mathcal{L})$$

$$(\mathcal{E},\mathcal{S},\mathcal{S}^{\mathrm{MS}},\mathcal{P}\cup^{\#}\{P|Q\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E},\mathcal{S},\mathcal{S}^{\mathrm{MS}},\mathcal{P}\cup^{\#}\{P,Q\},\sigma,\mathcal{L})$$

$$(\mathcal{E},\mathcal{S},\mathcal{S}^{\mathrm{MS}},\mathcal{P}\cup^{\#}\{!P\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E},\mathcal{S},\mathcal{S}^{\mathrm{MS}},\mathcal{P}\cup^{\#}\{!P,P\},\sigma,\mathcal{L})$$

$$(\mathcal{E},\mathcal{S},\mathcal{S}^{\mathrm{MS}},\mathcal{P}\cup^{\#}\{\nu a;P\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E}\cup\{a'\},\mathcal{S},\mathcal{S}^{\mathrm{MS}},\mathcal{P}\cup^{\#}\{P\{a'/a\}\},\sigma,\mathcal{L}) \quad \text{if } a' \text{ is fresh}$$

$$(\mathcal{E},\mathcal{S},\mathcal{S}^{\mathrm{MS}},\mathcal{P},\sigma,\mathcal{L}) \xrightarrow{K(M)} (\mathcal{E},\mathcal{S},\mathcal{S}^{\mathrm{MS}},\mathcal{P},\sigma,\mathcal{L}) \quad \text{if } \nu\mathcal{E}.\sigma\vdash M$$

$$(\ldots,\mathcal{P}\cup^{\#}\{\mathrm{out}(M,N);P\},\sigma,\mathcal{L}) \xrightarrow{K(M)} (\mathcal{E},\mathcal{S},\mathcal{S}^{\mathrm{MS}},\mathcal{P}\cup^{\#}\{P\},\sigma\cup\{^{N}/_{x}\},\mathcal{L}) \quad \text{if } x \text{ is fresh and } \nu\mathcal{E}.\sigma\vdash M$$

$$(\ldots,\mathcal{P}\cup^{\#}\{\mathrm{in}(M,N);P\},\sigma,\mathcal{L}) \xrightarrow{K(\langle M,N\tau\rangle)} (\ldots,\mathcal{P}\cup^{\#}\{P\tau\},\sigma,\mathcal{L}) \quad \text{if } \exists\tau.\ \tau \text{ is grounding for } N,\nu\mathcal{E}.\sigma\vdash M,\nu\mathcal{E}.\sigma\vdash N\tau$$

$$(\ldots,\mathcal{P}\cup^{\#}\{\mathrm{out}(M,N);P,\ \mathrm{in}(M',N');Q\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E},\mathcal{S},\mathcal{S}^{\mathrm{MS}},\mathcal{P}\cup^{\#}\{P,Q\tau\},\sigma,\mathcal{L}) \quad \text{if } M=_E M' \text{ and } \exists\tau.\ N=_E N'\tau \text{ and } \tau \text{ grounding for } N'$$

$$(\ldots,\mathcal{P}\cup^{\#}\{\text{if } pr(M_1,\ldots,M_n) \text{ then } P \text{ else } Q\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E},\mathcal{S},\mathcal{S}^{\mathrm{MS}},\mathcal{P}\cup^{\#}\{P\},\sigma,\mathcal{L}) \quad \text{if } \phi_{pr}\{^{M_1}/_{x_1},\ldots,^{M_n}/_{x_n}\} \text{ is satisfied}$$

$$(\ldots,\mathcal{P}\cup^{\#}\{\text{if } pr(M_1,\ldots,M_n) \text{ then } P \text{ else } Q\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E},\mathcal{S},\mathcal{S}^{\mathrm{MS}},\mathcal{P}\cup^{\#}\{Q\},\sigma,\mathcal{L}) \quad \text{if } \phi_{pr}\{^{M_1}/_{x_1},\ldots,^{M_n}/_{x_n}\} \text{ is not satisfied}$$

$$(\ldots,\mathcal{P}\cup^{\#}\{\mathrm{event}(F);P\},\sigma,\mathcal{L}) \xrightarrow{F} (\mathcal{E},\mathcal{S},\mathcal{S}^{\mathrm{MS}},\mathcal{P}\cup\{P\},\sigma,\mathcal{L})$$

Operations on global state:

$$(\ldots,\mathcal{P}\cup^{\#}\{\mathrm{insert}\ M,N;P\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E},\mathcal{S}[M\mapsto N],\mathcal{S}^{\mathrm{MS}},\mathcal{P}\cup^{\#}\{P\},\sigma,\mathcal{L})$$

$$(\ldots,\mathcal{P}\cup^{\#}\{\mathrm{delete}\ M;P\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E},\mathcal{S}[M\mapsto\bot],\mathcal{S}^{\mathrm{MS}},\mathcal{P}\cup^{\#}\{P\},\sigma,\mathcal{L})$$

$$(\ldots,\mathcal{P}\cup^{\#}\{\mathrm{lookup}\ M \text{ as } x \text{ in } P \text{ else } Q\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E},\mathcal{S},\mathcal{S}^{\mathrm{MS}},\mathcal{P}\cup^{\#}\{P\{V/x\}\},\sigma,\mathcal{L}) \quad \text{if } \mathcal{S}(N)=_E V \text{ is defined and } N=_E M$$

$$(\ldots,\mathcal{P}\cup^{\#}\{\mathrm{lookup}\ M \text{ as } x \text{ in } P \text{ else } Q\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E},\mathcal{S},\mathcal{S}^{\mathrm{MS}},\mathcal{P}\cup^{\#}\{Q\},\sigma,\mathcal{L}) \quad \text{if } \mathcal{S}(N) \text{ is undefined for all } N=_E M$$

$$(\ldots,\mathcal{P}\cup^{\#}\{\mathrm{lock}\ M;P\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E},\mathcal{S},\mathcal{S}^{\mathrm{MS}},\mathcal{P}\cup^{\#}\{P\},\sigma,\mathcal{L}\cup\{M\}) \quad \text{if } M\not\in_E\mathcal{L}$$

$$(\ldots,\mathcal{P}\cup^{\#}\{\mathrm{unlock}\ M;P\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E},\mathcal{S},\mathcal{S}^{\mathrm{MS}},\mathcal{P}\cup^{\#}\{P\},\sigma,\mathcal{L}\setminus\{M'\colon M'=_E M\})$$

$$(\ldots,\mathcal{P}\cup^{\#}\{[l\ \text{-}[\ a]\!\!\rightarrow\ r];\ P\},\sigma,\mathcal{L}) \xrightarrow{a'} (\mathcal{E},\mathcal{S},\mathcal{S}^{\mathrm{MS}}\setminus \mathit{lfacts}(l')\cup^{\#} r',\mathcal{P}\cup^{\#}\{P\tau\},\sigma,\mathcal{L})$$
$$\text{if } \exists\tau,l',a',r'.\tau \text{ grounding for } l\ \text{-}[\ a]\!\!\rightarrow\ r, l'\ \text{-}[\ a']\!\!\rightarrow\ r' =_E (l\ \text{-}[\ a]\!\!\rightarrow\ r)\tau,\ \mathit{lfacts}(l')\subseteq^{\#}\mathcal{S}^{\mathrm{MS}},\mathit{pfacts}(l')\subset\mathcal{S}^{\mathrm{MS}}$$

Fig. 3: Operational semantics

Definition 5 (Validity, satisfiability). *Let $Tr \subseteq (\mathcal{P}(\mathcal{G}))^*$ be a set of traces. A trace formula φ is said to be* valid *for Tr, written $Tr \models^\forall \varphi$, if for any trace $tr \in Tr$ and any valuation θ we have that $(tr, \theta) \models \varphi$.*

A trace formula φ is said to be satisfiable *for Tr, written $Tr \models^\exists \varphi$, if there exist a trace $tr \in Tr$ and a valuation θ such that $(tr, \theta) \models \varphi$.*

Note that $Tr \models^\forall \varphi$ iff $Tr \not\models^\exists \neg\varphi$. Given a ground process P we say that φ is valid, written $P \models^\forall \varphi$, if $traces^{pi}(P) \models^\forall \varphi$, and that φ is satisfied in P, written $P \models^\exists \varphi$, if $traces^{pi}(P) \models^\exists \varphi$.

4 Model

In this section, we will introduce our model of a PKCS#11 token. The complete code is available at `http://sapic.gforge.inria.fr/pkcs11templates.zip`and in Appendix B. We consider a security device that allows the creation of keys in its secure memory. The user can access the device via an API. If he creates a key, he obtains a handle, which he can use to let the device perform operations on his behalf. For each handle the device also stores a list of attributes, which define what operations are permitted for this handle. The goal is that the user can never gain knowledge of the key, as the user's machine might be compromised. We model the device by the following process (we use $\mathsf{out}(m)$ as a shortcut for $\mathsf{out}(c, m)$ for a public channel c):

$$P_{init}; !(P_{create} \mid P_{dec} \mid P_{enc} \mid P_{wrap} \mid P_{unwrap} \mid P_{get\_keyval}), \text{ where}$$

$P_{init} :=$
/* wrap, unwrap, enc, dec, sens, extr, trus, wwt, wt , ut */
insert ⟨'template', 'trusted'⟩,
⟨'on', 'on', 'off', 'off', 'on', 'on', 'on', 'on', 'usage', 'usage'⟩;
insert ⟨'template', 'usage'⟩,
⟨'off', 'off', 'on', 'on', 'on', 'on', 'off', 'on', 'undef', 'undef'⟩;
insert ⟨'template', 'untrusted'⟩,
⟨'off', 'off', 'on', 'on', 'off', 'on', 'off', 'off', 'undef', 'undef'⟩;

This sets up the templates before starting operation in the replicated process. Key-generation is handled by P_{create}:

$P_{create} :=$ in(⟨'create',*atts,ptr*⟩);
lock 'device';
ν *h*; ν *k*;
lookup ⟨'template', *ptr*⟩ as *templ* in
if *permits*(*attwrap*(*templ*), [...], *attut*(*templ*),
attwrap(*atts*), [...], *attut*(*atts*)) then
event *NewKey*(*h,k, attsens* (*atts*));
insert ⟨'obj', *h*⟩, ⟨*k*, *atts* ⟩;
event *WrapKey*(*h,k,attwrap*(*atts*)); event *DecKey*(*h,k, attdec* (*atts*));
event *EncKey*(*h,k, attenc* (*atts*)); event *UnwrapKey*(*h,k,attunwrap*(*atts*));
out(*h*) ;
unlock 'device'

Upon reception of a key generation request with a list of attributes and a pointer to a template, the device is locked. Then, the device creates a new handle h and a key k. The pointer is retrieved from the database P_{init} has written to. The functions $attwrap$, to $attut$ are simple deconstructors, $attwrap$, for example, extracts the first element from a list of 10 attributes (see Tab. 1). The predicate $permits$ compares the attributes given by the adversary with the attributes stored with the template. In subsection 2.2, we have argued that templates should determine the key's attributes, hence $permits$ is true if and only if the attributes including the pointers to the templates match exactly. To obtain a more permissive modelling, this predicate can be altered, e. g., to allow for certain attributes to be changed.

The creation of keys is logged as an event $\mathrm{NewKey}(h, k, attsens(atts))$. If the third argument is 'on', this key is sensitive, i. e., secrecy needs to be preserved. Events are used to state security properties and helping lemmas. Next, the device stores the key that belongs to the handle by associating the pair $\langle$'key', $h\rangle$ to the value of the key k and the attributes. The events $\mathrm{WrapKey}$ to $\mathrm{UnwrapKey}$ are used to refer to the attributes of keys in helping lemmas and otherwise irrelevant. Finally, the handle is output and the device unlocked.

Remark 1. The predicate $permits$ compares the attributes, including the pointers wt and ut, literally. Since our policy only accepts pointers to templates created by P_{init}, and since those are distinct, this is without loss of generality. Furthermore, the adversary has to provide the pointer to the template, which is without loss of generality, too, since the pointers are of sort pub.

If a handle has the 'dec' attribute set, it can be used for decryption:

P_{dec} := in($\langle$*h,senc(m,k)*$\rangle$);
lock 'device';
lookup $\langle$'obj', *h*$\rangle$ as *v* in
  if *can_decrypt*(*attwrap*(*tem*(*v*)), [...], *attut*(*tem*(*v*))) then
    if *key*(*v*)=*k* then
      event *DecUsing*(*k,m*); out(*m*); unlock 'device'

The lookup stores the value associated to $\langle$'obj', $h\rangle$ in v. The predicate $can\_decrypt$ is satisfied, iff. the fourth argument, $attdec(tem(v))$ equals 'on'. The function symbol tem extracts the second element of a pair, it is defined by the equation $tem(\langle k, t\rangle) = t$. Similarly, $key(\langle k, t\rangle) = k$. If the key stored with this handle matches the key used to generate the encryption, the plain-text is output and the device unlocked.

If a key has the 'wrap' attribute set, it can be used to encrypt the value of a second key:

P_{wrap} := in($\langle$*h1,h2*$\rangle$);
lock 'device';
lookup $\langle$'obj', *h1*$\rangle$ as *v1* in
  lookup $\langle$'obj', *h2*$\rangle$ as *v2* in
    if *can_wrap*(*attwrap*(*tem*(*v1*)),[...], *attut*(*tem*(*v1*)),
                attwrap(*tem*(*v2*)), [...], *attut*(*tem*(*v2*))) then
      lookup $\langle$'template', *attwt*(*tem*(*v1*))$\rangle$ as *wt* in

```
if permits( attwrap(wt), [...], attut(wt),
            attwrap(tem(v2)), [...], attut(tem(v2))) then
event Wrap(key(v1),key(v2));
out(senc(key(v2), key(v1)));
unlock 'device'
```

Given the two handles h_1 and h_2, the corresponding keys and attributes are retrieved. The predicate can_wrap is defined in Example 2 on page 10. If the template referred to by the entry 'wt' in the first key's attribute list permits wrapping the second key, i. e., it is equal to the second handle's attributes, then the encryption is output. The complete model including P_{enc}, P_{unwrap} and P_{get_keyval} can be found in Appendix B.

Limitations While being more detailed than previous works in terms of the attributes and commands supported, our model does have the following limitations: *Integrity of wrappings.* In our model, a wrapping can only be imported if it is an encryption of some message with a key that is on the device. In reality, this integrity property cannot be given. There are techniques that allow to account for malleability of cypher-texts in the symbolic model [3]. The current draft for version 2.40 indicates that authenticated encryption might be part of the future version of this standard [21] . *Multiple tokens.* We currently model a single token. By encapsulating the current process P in $\nu device\colon pub.P$ and prepending $device$ to database keys, this situation could be modelled. *Copying keys, asymmetric keys, key derivation.* C_CopyObject, support for asymmetric keys and key-derivation are not modelled for various reasons explained in Section 2.1. While we conjecture copying objects could be enabled, and asymmetric keys as well as key-derivation keys be allowed as 'usage' keys, this is missing in the current model (as opposed to other related work [11, 7]).

5 Security Results for the Proposed Policy

The main security result is the secrecy of keys generated on the device that have been marked 'sensitive' upon creation, in our case, keys created with the templates 'trusted' or 'usage'. This is expressed by the following trace formula:

$$\neg(\exists h, k\colon msg,\ i, j\colon temp.\ \mathrm{NewKey}(h, k, \text{'on'})@i \wedge \mathrm{K}(k)@j)$$

The action NewKey refers to the event in the key-generation process P_{create}, which we introduced in Section 4. If the third argument is 'on', k has been created 'sensitive'. To derive the result, we have defined 9 helping lemmas, four of which are rather trivial, but help speeding up the proof. The first, dec_limits, establishes typing invariants, most importantly, it states that decrypt is not useful to the adversary: Any message obtained by decryption was either known to the adversary in advance, or a key that was created 'sensitive' or imported was leaked, or there was some key that had the attributes 'wrap' and 'dec' set at different points in time. The following four lemmas state that given the templates it is, e. g., not possible to create a key with 'trusted' as wrapping template. The lemma bad_keys states that a key that was created by unwrapping must earlier have been created on the device, unless something bad happened, i. e., either a sensitive

name	interaction (w/o heur.)	interaction (w/ heur.)	proof size (w/o heur.)	proof size (w/ heur.)
dec_limits	11	0	3394	2235
trusted_as_ut_impossible	0	0	4	4
untrusted_as_ut_impossible	0	0	4	4
untrusted_as_wt_impossible	0	0	4	4
trusted_as_wt_impossible	0	0	4	4
bad_keys	0	0	2988	683
no_key_is_wrap_and_dec_[..]	15	0	1177	2396
no_key_is_enc_and_unwrap	29	0	2669	352
cannot_obtain_key_ind	6	0	7306	14598
cannot_obtain_key	0	0	0	0

Table 4: Evaluation of our proof method with and without the model-specific heuristics, broken down into lemmas. Interaction is measured in terms of mouse clicks. The user choses the next proof goal out of a list in a web interface. The value was determined by counting occurrences of the keyword 'solve' in the file. The proof size is the number of case distinctions.

key leaked, or a key had wrap and dec, or unwrap and enc set before. The latter two conflicts are known to cause attacks [8]. The following lemma no_key_is_wrap_and_-dec says that the first conflict can only occur if the second occurred before, or a sensitive key was leaked. Subsequently, it is shown the second conflict cannot occur, unless a key was leaked. The lemma cannot_obtain_key_ind is an inductive version of the security property cannot_obtain_key.

User intervention is necessary to find the proof when the tamarin prover is used with heuristics adapted to general Sapic output. They differ from the standard 'smart' heuristic only in that the actions corresponding to unlock operations and premises corresponding to the previous state in the execution are prioritized. Using a model-specific heuristic, it is possible to find the proof automatically. This heuristic prioritizes the resolution of insert operations with 'template' in the first argument (thereby moving the case distinction about which templates is used upwards in the proof tree) and deprioritizes the adversary's deduction of handles (as they are public anyway). With this heuristic, the complete proof is found within half an hour on a computation server with 24 Intel Xeon 2.67GHz cores and 50GB RAM. Experiments on desktop machines are planned. See Table 4 for more details. The manual part of the proof, transcripts of the complete proof and the tools used are available at `http://sapic.gforge.inria.fr/pkcs11templates.zip`.

6 Conclusion and Future work

We have investigated a new method of verifying key secrecy in PKCS#11 configurations following the protocol verification approach. It requires manual effort in defining helping lemmas, but overcomes the need for a model-specific approximation techniques and models PKCS#11 in a precise and intuitive manner. In particular, our model supports features that have been added in version 2.20 of the standard. The upcoming [21]

version 2.40 is available for public review and will support encryption with authenticated data, a mechanism that has long been requested and has the potential to remove the restriction to three-level policies outlined in Section 2.2 and allow for new policies. Future work on version 2.40 will benefit from the fact that our verification technique applies without model-specific approximation. The second contribution is a secure configuration of PKCS#11 that permits wrapping and unwrapping for backup purposes.

Besides adapting our model to the case of multiple tokens in the network, we plan to increase the degree of automation by refining the model-specific heuristic for the tamarin prover's constraint solving algorithm. The current heuristic is rather simple, but may be generalized. We stress that completeness and soundness of the tool chain are independent of the heuristic used, suggesting a tool chain without approximation but adapted heuristics as an alternative to ad-hoc approximations that does not necessarily provide decidability, but is more flexible with regard to extensions and less reliant on hand-written proofs.

Acknowledgements This work was supported by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE.

References

[1] Martín Abadi and Cédric Fournet. "Mobile Values, New Names, and Secure Communication". In: *POPL 2001*. ACM Press, 2001.

[2] Pedro Adão, Riccardo Focardi, and Flaminia L. Luccio. "Type-Based Analysis of Generic Key Management APIs". In: *CSF 2013*. IEEE, 2013, pp. 97–111.

[3] Naveed Ahmed, Christian D. Jensen, and Erik Zenner. "Towards Symbolic Encryption Schemes". English. In: *ESORICS 2012*. Vol. 7459. LNCS. Springer, 2012.

[4] Romain Bardou et al. "Efficient Padding Oracle Attacks on Cryptographic Hardware". In: *CRYPTO 2012*. Vol. 7417. LNCS. Springer, 2012.

[5] Mike Bond and Ross Anderson. "API level attacks on embedded systems". In: *IEEE Computer Magazine* 34.10 (2001).

[6] Matteo Bortolozzo et al. "Attacking and Fixing PKCS#11 Security Tokens". In: *CCS 2010*. ACM Press, 2010.

[7] Matteo Centenaro, Riccardo Focardi, and Flaminia L. Luccio. "Type-based analysis of key management in PKCS#11 cryptographic devices". In: *Journal of Computer Security* 21.6 (2013).

[8] Jolyon Clulow. "On the Security of PKCS #11". English. In: *CHES 2003*. Ed. by Colin D. Walter, Çetin K. Koç, and Christof Paar. Springer, 2003.

[9] V. Cortier, G. Keighren, and G. Steel. "Automatic Analysis of the Security of XOR-based Key Management Schemes". In: *TACAS 2007*. Vol. 4424. LNCS. Springer, 2007.

[10] Véronique Cortier, Graham Steel, and Cyrille Wiedling. "Revoke and let live: a secure key revocation API for cryptographic devices". In: *CCS 2012*. ACM, 2012.

[11] Stéphanie Delaune, Steve Kremer, and Graham Steel. "Formal Analysis of PKCS#11 and Proprietary Extensions". In: *Journal of Computer Security* 18.6 (Nov. 2010).
[12] N. Durgin et al. "Undecidability of Bounded Security Protocols". In: *Workshop on Formal Methods and Security Protocols*. IEEE, 1999.
[13] Sibylle Fröschle and Nils Sommer. "Concepts and Proofs for Configuring PKCS#11". In: *FAST 2011*. Vol. 7140. LNCS. Leuven, Belgium: Springer, 2012.
[14] Sibylle Fröschle and Graham Steel. "Analysing PKCS#11 Key Management APIs with Unbounded Fresh Data". In: *Joint Workshop on Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (ARSPA-WITS'09)*. Vol. 5511. LNCS. Springer, 2009.
[15] Sibylle B. Fröschle and Nils Sommer. "Reasoning with Past to Prove PKCS#11 Keys Secure". In: *FAST 2010*. Vol. 6561. LNCS. Springer, 2010.
[16] Sibylle B. Fröschle and Nils Sommer. *When is a PKCS#11 configuration secure?* Tech. rep. Reports of SFB/TR 14 AVACS 82, SFB/TR 14 AVACS, 2011. URL: https://vhome.offis.de/sibyllef/cryptokireport.pdf.
[17] Steve Kremer and Robert Künnemann. "Automated analysis of security protocols with global state". In: *Security and Privacy*. IEEE Computer Society, 2014.
[18] Steve Kremer, Robert Künnemann, and Graham Steel. "Universally Composable Key-Management". In: *ESORICS 2013*. Vol. 8134. LNCS. Springer, 2013.
[19] Steve Kremer, Graham Steel, and Bogdan Warinschi. "Security for Key Management Interfaces". In: *CSF 2011*. IEEE Computer Society, 2011, pp. 66–82.
[20] Dennis Longley and Simon Rigby. "An Automatic Search for Security Flaws in Key Management Schemes". In: *Computers and Security* 11.1 (Mar. 1992).
[21] *PKCS #11 Cryptographic Token Interface Base Specification Version 2.40, Committee Specification 01*. OASIS Open. Sept. 2014. URL: http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/cs01/pkcs11-base-v2.40-cs01.html.
[22] *PKCS #11: Cryptographic Token Interface Standard.* RSA Security Inc. v2.20, June 2004.
[23] Benedikt Schmidt et al. "Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties". In: *CSF 2012*. IEEE, 2012.

A Modifications to the proof of correctness in Kremer et.al.

We employ an slightly extended version of the sapic calculus presented by Kremer et.al. [17], which permits the use of a class of predicates in if-statements. These and other extensions are unpublished so far, but for the sake of completeness, we will sketch the changes necessary to preserve the correctness result, in particular the translation procedure and the proofs to Lemma 10 and Lemma 12.

Definition 12 (Reserved variables and facts). *The set of reserved facts $\mathcal{F}_{res}$ additionally contains facts Pred_pr $(t_1, \ldots, t_n)$ and Pred_not_pr $(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n \in \mathcal{T}$, and $pr \in \Sigma_{pred}$.*

Definition 15. *The rule for the if-case is substituted by the following one:*

$$\begin{aligned}[\![\textit{if } pr(M_1,\ldots,M_k) \textit{ then } P \textit{ else } Q, p, \tilde{x}]\!] = \\ [\mathsf{state}_p(\tilde{x})] \dashv[\ \textit{Pred}_{pr}(M,N)]\rightarrow [\mathsf{state}_{p\cdot 1}(\tilde{x})], \\ [\mathsf{state}_p(\tilde{x})] \dashv[\ \textit{Pred\_not}_{pr}(M,N)]\rightarrow [\mathsf{state}_{p\cdot 2}(\tilde{x})], \\ \cup\, [\![P, p\cdot 1, \tilde{x}]\!] \cup [\![Q, p\cdot 2, \tilde{x}]\!]\end{aligned}$$

Definition 16. *We define* $\alpha := \alpha_{init} \wedge \alpha_{pred} \wedge \alpha_{noteq} \wedge \alpha_{in} \wedge \alpha_{notin} \wedge \alpha_{lock} \wedge \alpha_{inev}$ *and* α_{pred}*, as follows. Other definitions are left unaltered.*

$$\begin{aligned}\alpha_{pred} := \\ \bigwedge_{pr\in\Sigma_{pred}} \{\forall x_1,\ldots,x_k,i. \quad \textit{Pred}_{pr}(x_1,\ldots,x_k)@i \Longrightarrow \phi_{pr} \mid pr \textit{ is of arity } k\}\wedge \\ \bigwedge_{pr\in\Sigma_{pred}} \{\forall x_1,\ldots,x_k,i. \quad \textit{Pred\_not}_{pr}(x_1,\ldots,x_k)@i \Rightarrow \neg(\phi_{pr}) \mid pr \textit{ of arity } k\}\end{aligned}$$

Lemma 11. *We alter the two cases pertaining to if as follows:*

Proof. **Case** $(\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}^{\mathrm{MS}}_{n-1}, \mathcal{P}_{n-1} = \mathcal{P}' \cup \{\,\mathsf{if}\ pr(t_1,\ldots,t_l)\ \mathsf{then}\ Q\ \mathsf{else}\ Q'\,\}, \sigma_{n-1}, \mathcal{L}_{n-1}) \rightarrow (\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}^{\mathrm{MS}}_{n-1}, \mathcal{P}' \cup \{\,Q\,\}, \sigma_{n-1}, \mathcal{L}_{n-1})$. This step requires that $\sigma_{pr}\,\{{}^{t_1}/_{x_1},\ldots,\ {}^{t_l}/_{x_l}\}$ is satisfied. By induction hypothesis $\mathcal{P}_{n-1} \leftrightarrow_P S_{n'}$. Let p and $\tilde{t}$ be such that

$$\mathsf{if}\ pr(t_1,\ldots,t_l)\ \mathsf{then}\ Q\ \mathsf{else}\ Q' \leftrightarrow_P \mathsf{state}_p(\tilde{t}).$$

By Def. 20 there is a $ri \in \mathit{ginsts}([\![P]\!]_{=p})$ such that $\mathsf{state}_p(\tilde{t})$ is part of its premise. By definition of $[\![P]\!]_{=p}$, we can choose $ri = [\mathsf{state}_p(\tilde{t})] \dashv[\ \mathrm{Pred}_{pr}(t_1,\ldots,t_l)] \rightarrow [\mathsf{state}_{p\cdot 1}(\tilde{t})]$. We can extend the previous execution by one step using ri, therefore:

$$\emptyset \xrightarrow{F_1}_{[\![P]\!]} S_1 \xrightarrow{F_2}_{[\![P]\!]} \cdots \xrightarrow{F_{n'}}_{[\![P]\!]} S_{n'} \xrightarrow{\mathrm{Pred}_{pr}(t_1,\ldots,t_l)}_{[\![P]\!]} S_{n'+1} \in \mathit{exec}^{msr}([\![P]\!])$$

with $S_{n'+1} = \{\,S_{n'} \setminus^{\#} \{\,\mathsf{state}_p(\tilde{t})\,\}^{\#} \cup^{\#} \{\,\mathsf{state}_{p\cdot 1}(\tilde{t})\,\}^{\#}\,\}$. It is left to show that Conditions 1 to 8 hold for n. The last step is labelled $F_{f(n)} = \mathrm{Pred}_{pr}(t_1,\ldots,t_l)$. As $\sigma_{pr}\,\{{}^{t_1}/_{x_1},\ldots,{}^{t_l}/_{x_l}\}$ is satisfied, Condition 7 holds, in particular, α_{pred} is not violated. Since Pred_{pr} is reserved, Condition 8 holds as well.

As before, since $\mathcal{P}_n = \mathcal{P}_{n-1} \setminus^{\#} \{\,\mathsf{if}\ pr(t_1,\ldots,t_l)\ \mathsf{then}\ Q\ \mathsf{else}\ Q'\,\} \cup^{\#} \{Q\}$ and $\{Q\} \leftrightarrow \{\mathsf{state}_{p\cdot 1}(\tilde{t}, a)\}$ (by definition of the translation), we have that $\mathcal{P}_n \leftrightarrow_P S_{f(n)}$, and therefore Condition 4 holds. Condition 1, Condition 2, Condition 3, Condition 5 and Condition 6 hold trivially.

Case $(\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}^{\mathrm{MS}}_{n-1}, \mathcal{P}_{n-1} = \mathcal{P}' \cup \{\,\mathsf{if}\ pr(t_1,\ldots,t_l)\ \mathsf{then}\ Q\ \mathsf{else}\ Q'\,\}, \sigma_{n-1}, \mathcal{L}_{n-1}) \rightarrow (\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}^{\mathrm{MS}}_{n-1}, \mathcal{P}' \cup \{\,Q'\,\}, \sigma_{n-1}, \mathcal{L}_{n-1})$. This step requires that the predicate $\sigma_{pr}\,\{{}^{t_1}/_{x_1},\ldots,{}^{t_l}/_{x_l}\}$ is not satisfied. This proof step is similar to the previous case, except ri is chosen to be

$$[\mathsf{state}_p(\tilde{t})] \dashv[\ \mathrm{Pred\_not}_{pr}(t_1,\ldots,t_l)]\rightarrow [\mathsf{state}_{p\cdot 2}(\tilde{t})].$$

The condition in α_{noteq} holds since $\sigma_{pr}\,\{{}^{t_1}/_{x_1},\ldots,{}^{t_l}/_{x_l}\}$ is not satisfied, and thus, by definition of the satisfaction relation, $\neg\sigma_{pr}\,\{{}^{t_1}/_{x_1},\ldots,{}^{t_l}/_{x_l}\}$ is satisfied. □

Lemma 13. *We alter the two cases pertaining to if as follows:*

Proof. **Case** $ri = [\mathsf{state}_p(\tilde{t})] \dashv[\ \mathrm{Pred}_{pr}(t_1, \ldots, t_l)] \rightarrow [\mathsf{state}_{p\cdot 1}(\tilde{t})]$ (for some p, $t_1, \ldots, t_l \in \mathcal{M}$ and $\tilde{t}$). By induction hypothesis, we have $\mathcal{P}_{n'} \leftrightsquigarrow_P S_m$, and thus, as previously established, $\mathcal{P}_{n'} \leftrightsquigarrow_P S_{n-1}$. Let $Q \in^{\#} \mathcal{P}_{n'}$ such that $Q \leftrightsquigarrow_P \mathsf{state}_p(\tilde{t})$. Let θ be a grounding substitution for $\mathsf{state}_p(\tilde{x}) \in prems([\![P]\!]_{=p})$ such that $\tilde{t} = \tilde{x}\theta$. Then θ induces a substitution τ and a bijective renaming ρ for fresh, but not bound names (in Q) such that $P|_p\tau\rho = Q$ (see Def. 22).

From the form of the rule R, and since $Q = P|_p\tau\rho$, we can deduce that

$$Q = \mathsf{if}\ pr(t_1, \ldots, t_l)\ \mathsf{then}\ Q_1\ \mathsf{else}\ Q_2$$

for a process $Q' = P|_{p\cdot 1}\tau\rho$.

Since $[E_1, \ldots, E_n] \vDash \alpha$, and thus $[E_1, \ldots, E_m] \vDash \alpha_{pred}$, $\sigma_{pr}\,\{^{t_1}/_{x_1}, \ldots, ^{t_l}/_{x_l}\}$ is satisfied. We therefore chose the following transition:

$$\cdots \xrightarrow{F'_n} (\mathcal{E}_{n'}, \mathcal{S}_{n'}, \mathcal{S}^{\mathrm{MS}}_{n'}, \mathcal{P}_{n'}, \sigma_{n'}, \mathcal{L}_{n'}) \rightarrow (\mathcal{E}_{n'+1}, \mathcal{S}_{n'+1}, \mathcal{S}^{\mathrm{MS}}_{n'+1}, \mathcal{P}_{n'+1}, \sigma_{n'+1}, \mathcal{L}_{n'+1})$$

with $\mathcal{E}_{n'+1} = \mathcal{E}_{n'}$, $\mathcal{S}_{n'+1} = \mathcal{S}_{n'}$, $\mathcal{S}^{\mathrm{MS}}_{n'+1} = \mathcal{S}^{\mathrm{MS}}_{n'}$, $\mathcal{P}_{n'+1} = \mathcal{P}_{n'} \setminus^{\#} \{\, \mathsf{if}\ pr(t_1, \ldots, t_l)\ \mathsf{then}\ Q_1 \mathsf{else}\ Q'_2 \,\}^{\#} \cup^{\#} \{\, Q_1 \,\}^{\#}$, $\sigma_{n'+1} = \sigma_{n'}$ and $\mathcal{L}_{n'+1} = \mathcal{L}_{n'}$.

We define f as on the first page in the original proof. Therefore, Conditions 1 to 8 hold for $i < n - 1$. It is left to show that Conditions 1 to 8 hold for n.

By definition of $[\![P]\!]$ and $[\![P]\!]_{=p}$, we have that $Q_1 \leftrightarrow \mathsf{state}_{p\cdot 1}(\tilde{t})$. Therefore, and since $\mathsf{if}\ pr(t_1, \ldots, t_l)\ \mathsf{then}\ Q_1\ \mathsf{else}\ Q_2 \leftrightarrow \mathsf{state}_p(\tilde{t})$, we have that $\mathcal{P}_{n'+1} = \mathcal{P}_{n'} \setminus^{\#} \{\, \mathsf{if}\ pr(t_1, \ldots, t_l)\ \mathsf{then}\ Q_1\ \mathsf{else}\ Q_2 \,\}^{\#} \cup^{\#} \{\, Q_1 \,\}^{\#}$, and $S_n = S_{n-1} \setminus^{\#} \{\, \mathsf{state}_p(\tilde{t}) \,\}^{\#} \cup^{\#} \{\, \mathsf{state}_{p\cdot 1}(\tilde{t}) \,\}^{\#}$, Condition 4 holds. Conditions 1, 2, 3, 5, 6 and 7 hold trivially.

Case $ri = [\mathsf{state}_p(\tilde{t})] \dashv[\ \mathrm{Pred\_not}_{pr}(t_1, \ldots, t_l)] \rightarrow [\mathsf{state}_{p\cdot 1}(\tilde{t})]$ (for some $p, \tilde{t}$ and $t_1, \ldots, t_l \in \mathcal{M}$). In this case, the proof is almost the same as in the previous case, except that the predicate $\neg\sigma_{pr}\,\{^{t_1}/_{x_1}, \ldots, ^{t_l}/_{x_l}\}$ is satisfied, and thus $\sigma_{pr}\,\{^{t_1}/_{x_1}, \ldots, ^{t_l}/_{x_l}\}$ is not satisfied, Q_2 is chosen instead of Q_1 and and $S_n = S_{n-1} \setminus^{\#} \{\, \mathsf{state}_p(\tilde{t}) \,\}^{\#} \cup^{\#} \{\, \mathsf{state}_{p\cdot 2}(\tilde{t}) \,\}^{\#}$.

□

B The complete model

```
theory PKCS11TemplatePolicy

/*
 * Protocol:      PKCS#11
 * Modeler:       Robert Kunnemann
 * Date:          Oct 2014
 *
 * Status:        working
 */
 /*
 * PKCS#11 Configuration with three  templates:
```

```
 *       − one for  trusted  keys  that  can backup other  trusted  keys
 *       − one for  trusted  keys  that  can backup other  non−trusted keys
 *       − one non−trusted keys ,  that  can¬backup anything
 */

begin

builtins: symmetric-encryption

functions: key/1, attwrap/1, attunwrap/1, attenc/1, attdec/1,
    attsens/1,
attextr/1, atttrus/1, attwwt/1, attwt/1, attut/1, key/1, tem/1

equations:

 /* extract  attributes */
attwrap (<wrap, unwrap, enc, dec, sens, extr, trus, wwt, wt, ut
    >) = wrap,
attunwrap (<wrap, unwrap, enc, dec, sens, extr, trus, wwt, wt,
    ut>) = unwrap,
attenc (<wrap, unwrap, enc, dec, sens, extr, trus, wwt, wt, ut>)
     = enc,
attdec (<wrap, unwrap, enc, dec, sens, extr, trus, wwt, wt, ut>)
     = dec,
attsens (<wrap, unwrap, enc, dec, sens, extr, trus, wwt, wt, ut
    >) = sens,
attextr (<wrap, unwrap, enc, dec, sens, extr, trus, wwt, wt, ut
    >) = extr,
atttrus (<wrap, unwrap, enc, dec, sens, extr, trus, wwt, wt, ut
    >) = trus,
attwwt (<wrap, unwrap, enc, dec, sens, extr, trus, wwt, wt, ut>)
     = wwt,
attwt (<wrap, unwrap, enc, dec, sens, extr, trus, wwt, wt, ut>)
    = wt,
attut (<wrap, unwrap, enc, dec, sens, extr, trus, wwt, wt, ut>)
    = ut,
key (<k,templ>) = k,
tem (<k,templ>) = templ

predicates:
can_encrypt(wrap, unwrap, enc, dec, sens, extr, trus, wwt, wt,
    ut) <=> enc='on',
can_decrypt(wrap, unwrap, enc, dec, sens, extr, trus, wwt, wt,
    ut) <=> dec='on',
can_wrap(
wrap1, unwrap1, enc1, dec1, sens1, extr1, trus1, wwt1, wt1, ut1,
wrap2, unwrap2, enc2, dec2, sens2, extr2, trus2, wwt2, wt2, ut2
) <=> (wrap1='on' ∧extr2='on') &
( (wwt2='off')
 ∨ ( wwt2='on' ∧trus1='on'))
```

```
,
can_unwrap(wrap, unwrap, enc, dec, sens, extr, trus, wwt, wt, ut
    ) <=> unwrap='on',
can_get_keyvalue(wrap, unwrap, enc, dec, sens, extr, trus, wwt,
    wt, ut) <=> sens='off',
permits(
t_wrap, t_unwrap, t_enc, t_dec, t_sens, t_extr, t_trus, t_wwt,
    t_wt, t_ut,
wrap, unwrap, enc, dec, sens, extr, trus, wwt, wt, ut) <=>
// permissiveness can be alted by changing this predicate.
(t_wrap = wrap) &
(t_unwrap = unwrap) &
(t_enc = enc) &
(t_dec = dec) &
(t_sens = sens) &
(t_extr = extr) &
(t_trus = trus) &
(t_wwt = wwt) &
(t_wt = wt) &
(t_ut = ut)

let create = (in(<'create',atts,ptr>);
   lock 'device';
   νh; νk;
   // wrap, unwrap, enc, dec, sens, extr, trus, wwt, wt, ut
   lookup <'template',ptr> as templ in
      if permits(
      attwrap(templ), attunwrap(templ), attenc(templ), attdec(
          templ),
attsens(templ), attextr(templ), atttrus(templ), attwwt(templ),
attwt(templ), attut(templ),
      attwrap(atts), attunwrap(atts), attenc(atts), attdec(atts),
attsens(atts), attextr(atts), atttrus(atts), attwwt(atts),
attwt(atts), attut(atts)
      ) then
         event NewKey(h,k,attsens(atts));
         insert <'obj',h>, <k,atts>;
         event WrapKey(h,k,attwrap(atts));
         event DecKey(h,k,attdec(atts));
         event EncKey(h,k,attenc(atts));
         event UnwrapKey(h,k,attunwrap(atts));
         out(h) ;
         unlock 'device'
    )


let decrypt= (
   in(<h,senc(m,k)>);
   lock 'device';
   lookup <'obj',h> as v in
```

```
      if can_decrypt(
      attwrap(tem(v)), attunwrap(tem(v)), attenc(tem(v)), attdec(
          tem(v)),
attsens(tem(v)), attextr(tem(v)), atttrus(tem(v)), attwwt(tem(v)),
attwt(tem(v)), attut(tem(v))) then
            if key(v)=k then
               event DecUsing(k,m); out(m); unlock 'device'
      )

let encrypt = (in(<h,m>);
   lock 'device';
   lookup <'obj',h> as v in
      if can_encrypt(attwrap(tem(v)), attunwrap(tem(v)), attenc(
          tem(v)), attdec(tem(v)),
attsens(tem(v)), attextr(tem(v)), atttrus(tem(v)), attwwt(tem(v)),
attwt(tem(v)), attut(tem(v))) then
         event EncUsing(key(v),m); out(senc(m,key(v))); unlock '
             device'
      )

let wrap =
  ( in(<h1,h2>);
   lock 'device';
   lookup <'obj',h1> as v1 in
      lookup <'obj',h2> as v2 in
         if can_wrap(
attwrap(tem(v1)), attunwrap(tem(v1)), attenc(tem(v1)), attdec(tem(
    v1)),
attsens(tem(v1)), attextr(tem(v1)), atttrus(tem(v1)), attwwt(tem(
    v1)),
attwt(tem(v1)), attut(tem(v1)), attwrap(tem(v2)), attunwrap(tem(v2
    )),
attenc(tem(v2)), attdec(tem(v2)), attsens(tem(v2)), attextr(tem(v2
    )),
atttrus(tem(v2)), attwwt(tem(v2)), attwt(tem(v2)), attut(tem(v2)))
     then
            lookup <'template', attwt(tem(v1))> as wt in
               if permits(
                     attwrap(wt), attunwrap(wt), attenc(wt),
                         attdec(wt),
                     attsens(wt), attextr(wt), atttrus(wt),
                         attwwt(wt),
                     attwt(wt), attut(wt),
                     attwrap(tem(v2)), attunwrap(tem(v2)), attenc
                         (tem(v2)),
                     attdec(tem(v2)), attsens(tem(v2)), attextr(
                         tem(v2)),
                     atttrus(tem(v2)), attwwt(tem(v2)), attwt(tem(
                         v2)),
                     attut(tem(v2))
```

```
            ) then
               event Wrap(key(v1),key(v2));
               out(senc(key(v2),key(v1)));
               unlock 'device'
         )

let unwrap =
   ( in(<h,senc(m,k),atts>);
     lock 'device';
     lookup <'obj',h> as v in
       if can_unwrap(attwrap(tem(v)), attunwrap(tem(v)), attenc(tem
           (v)), attdec(tem(v)),
attsens(tem(v)), attextr(tem(v)), atttrus(tem(v)), attwwt(tem(v)),
attwt(tem(v)), attut(tem(v))) then
         if key(v)=k then
            lookup <'template', attut(tem(v))> as ut in
               if permits(
                     attwrap(ut), attunwrap(ut), attenc(ut),
                         attdec(ut),
                     attsens(ut), attextr(ut), atttrus(ut),
                         attwwt(ut),
                     attwt(ut), attut(ut),
                     attwrap(atts), attunwrap(atts), attenc(atts)
                         ,
                     attdec(atts), attsens(atts), attextr(atts),
                     atttrus(atts), attwwt(atts), attwt(atts),
                     attut(atts)

               ) then
                  ν h2;
                  insert <'obj',h2>, <m, atts >;
                  event Unwrapped(h2,m, atts);
                  event WrapKey(h2,m,attwrap(atts));
                  event DecKey(h2,m,attdec(atts));
                  event EncKey(h2,m,attenc(atts));
                  event UnwrapKey(h2,m,attunwrap(atts));
                  out(h2) ;
                  unlock 'device'
   )

let get_keyvalue= (in(h);
   lock 'device';
   lookup <'obj',h> as v in
      if can_get_keyvalue(
      attwrap(tem(v)), attunwrap(tem(v)), attenc(tem(v)), attdec(
          tem(v)),
attsens(tem(v)), attextr(tem(v)), atttrus(tem(v)), attwwt(tem(v)),
attwt(tem(v)), attut(tem(v))) then
         event GetKeyValue(key(v)); out(key(v)); unlock 'device'
      )
```

```
insert <'template','trusted'>,
//        wrap, unwrap, enc, dec, sens, extr, trus, wwt, wt, ut
       < 'on', 'on', 'off','off','on','on', 'on', 'on','usage','usage'>;
insert <'template','usage'>,
//       wrap, unwrap, enc, dec, sens, extr, trus, wwt, wt, ut
       <'off', 'off','on','on','on', 'on','off', 'on','undef','undef'>;
insert <'template','untrusted'>,
//       wrap, unwrap, enc, dec, sens, extr, trus, wwt, wt, ut
       <'off', 'off','on','on','off', 'on','off', 'off','undef','undef'>;
!(
  create
 | decrypt
 | encrypt
 | wrap
 | unwrap
 | get_keyvalue
)

/* Sanity tests */

/* // verified in 3 steps (w/ bounds) */
/* lemma can_create_key: //for sanity */
/*   exists-trace */
/*   "∃#t h k. NewKey(h,k)@t" */

/* // verified in 18 steps (w/ bounds) */
/* lemma can__wrap: //for sanity */
/*   exists-trace */
/*   "∃#t k1 k2 .Wrap(k1,k2)@t" */

/* // verified in 13 steps (w/ bounds) */
/* lemma can_enc: //for sanity */
/*   exists-trace */
/*   "∃#t k m . EncUsing(k,m)@t" */


/* // verified in 21 steps (w/ bounds) */
/* lemma can_dec: //for sanity */
/*   exists-trace */
/*   "∃#t k m . DecUsing(k,m)@t" */

/* // can be found manually */
/* lemma can_unwrap: //for sanity */
/*   exists-trace */
/*   "∃#t h k .Unwrapped(h,k)@t" */

lemma dec_limits[typing]:
  "(
```

```
          ∀k m #t1. DecUsing(k,m)@t1 ⟹
          ( ∃h2 k2 #t2 #t3 . NewKey(h2, k2, 'on')@t2 ∧ KU(k2)@t3 ∧t2<t1 ∧ t3<t1
              )
        ∨ ( ∃h2 #t2 #t3 #t4 . NewKey(h2, k, ' off ')@t2 ∧ KU(k)@t3 ∧KU(m)@t4 ∧
            t2<t1 ∧ t3<t1 ∧ t4<t1) //ν
  ∨ ( ∃#t2 . EncUsing(k,m)@t2 ∧t2<t1 )
        ∨ ( ∃h2 k2 #t2 #t3 a. Unwrapped(h2, k2, a)@t2 ∧ KU(k2)@t3 ∧t2<t1 ∧ t3<
            t1)
        ∨ ( ∃#t2 #t3 h1 h2 k2 . WrapKey(h2, k2,'on') @ t2 ∧ DecKey(h1,k2,'on') @
            t3
                                  ∧ t2<t1 ∧ t3<t1
          )
        )
        &
        // get_keyvalue
        (∀k #t2 . GetKeyValue(k)@t2 ⟹∃h #t1 . NewKey(h,k,'off')@t1 )
         ∧ /∗ object that can¬be created because there is ¬template that
              allows their creation ∗/
        (¬(∃h k wrap unwrap enc dec sens extr trus wwt wt #t.
        Insert ( <'obj', h>,
                    <k, wrap, unwrap, enc, dec, sens, extr, trus, wwt, wt, ' trusted '>
          ) @ t ))
        &
        (¬(∃h k wrap unwrap enc dec sens extr trus wwt wt #t.
        Insert ( <'obj', h>,
                    <k, wrap, unwrap, enc, dec, sens, extr, trus, wwt, wt, ' untrusted
                        '>
          ) @ t ))
        &
        (¬(∃h k wrap unwrap enc dec sens extr trus wwt ut #t.
        Insert ( <'obj', h>,
                    <k, wrap, unwrap, enc, dec, sens, extr, trus, wwt, ' trusted ', ut>
          ) @ t ))
        &
        (¬(∃h k wrap unwrap enc dec sens extr trus wwt ut #t.
        Insert ( <'obj', h>,
                    <k, wrap, unwrap, enc, dec, sens, extr, trus, wwt, ' untrusted ', ut
                        >
          ) @ t ))
        "

lemma trusted_as_ut_impossible[reuse]:
    "¬(∃h k wrap unwrap enc dec sens extr trus wwt wt #t.
        Insert ( <'obj', h>,
                    <k, wrap, unwrap, enc, dec, sens, extr, trus, wwt, wt, ' trusted '>
          ) @ t )"

lemma untrusted_as_ut_impossible[reuse]:
    "¬(∃h k wrap unwrap enc dec sens extr trus wwt wt #t.
```

```
        Insert ( <'obj ', h>,
                  <k, wrap, unwrap, enc, dec, sens, extr , trus , wwt, wt, ' untrusted
                      '>
          ) @ t )"

lemma untrusted_as_wt_impossible[reuse]:
    "¬( ∃h k wrap unwrap enc dec sens extr trus wwt ut #t.
        Insert ( <'obj ', h>,
                  <k, wrap, unwrap, enc, dec, sens, extr , trus , wwt, ' untrusted ', ut
                      >
          ) @ t )"

lemma trusted_as_wt_impossible[reuse]:
    "¬( ∃h k wrap unwrap enc dec sens extr trus wwt ut #t.
        Insert ( <'obj ', h>,
                  <k, wrap, unwrap, enc, dec, sens, extr , trus , wwt, ' trusted ', ut>
          ) @ t )"

lemma bad_keys [reuse, use_induction]:
      "
         ∀h2 k #t2 a . Unwrapped (h2,k,a)@t2 ⟹
               (∃h1 #t1 . NewKey (h1,k, 'on')@t1 ∧ t1<t2 )
           ∨ ( ∃h k2 #t1 #t0. NewKey(h, k2, 'on')@t0 ∧ KU(k2)@t1 ∧t1<t2 ∧ t0<
               t2)
           ∨ ( ∃#t0 #t1 h1 h2 k. WrapKey(h2, k,'on')@ t0 ∧ DecKey(h1,k,'on') @
               t1
               ∧ t0<t2 ∧ t1<t2 )
           ∨ ( ∃#t0 #t1 h1 h2 k. UnwrapKey(h2, k,'on')@ t0 ∧ EncKey(h1,k,'on')
               @ t1
               ∧ t0<t2 ∧ t1<t2 )
       "

lemma no_key_is_wrap_and_dec__or_unwrap_and_dec_ind[use_induction,
    reuse]:
       "
       (∀#t2 #t3 h1 h2 k . (DecKey(h1,k,'on') @ t2 ∧ WrapKey(h2, k,'on') @ t3)
       ⟹
         ( ∃h k2 #t1 #t0 . NewKey(h, k2, 'on')@t0 ∧ KU(k2)@t1
            &
           ( ( t1<t3 ∧ t0<t3)
             ∨ ( t1<t2 ∧ t0<t2) )
         )
           ∨ ( ∃#t0 #t1 h1 h2 k. UnwrapKey(h2, k,'on')@ t0 ∧ EncKey(h1,k,'on')
               @ t1
                  &(
                     ( t0<t2 ∧ t1<t2)
                    |( t0<t3 ∧ t1<t3)
                  ))
      )
```

```
       "

lemma no_key_is_enc_and_unwrap[use_induction,reuse]:
       "
       (∀ #t2 #t3 h1 h2 k .  (EncKey(h1,k,'on') @ t2 ∧ UnwrapKey(h2, k,'on') @ t3)
       ⟹
         ( ∃h k2 #t1 #t0 . NewKey(h, k2, 'on')@t0 ∧ KU(k2)@t1
             &
           (  ( t1<t3  ∧ t0<t3)
             ∨ ( t1<t2  ∧ t0<t2) )
         )
       )
       "

lemma cannot_obtain_key_ind[reuse,use_induction]:
  "¬(∃ #i #j  h k . NewKey(h,k,'on')@i ∧ KU(k) @j)"

lemma cannot_obtain_key:
  "¬(∃ #i #j  h k . NewKey(h,k,'on')@i ∧ K(k) @j)"

end
```